TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Elektrotechnik und Informationstechnik Institut für Automatisierungstechnik,
Professur für Prozessleittechnik / AG Systemverfahrenstechnik

S. Hensel, M. Graube, L. Urbas

METHODOLOGY FOR CONFLICT DETECTION AND RESOLUTION IN SEMANTIC REVISION CONTROL SYSTEMS

PLT-BERICHT 2016-08-A

Technische Universität Dresden
Fakultät Elektrotechnik und Informationstechnik
Institut für Automatisierungstechnik
Professur für Prozessleittechnik
01062 Dresden

Professur für PROZESS LEITTECHNIK

DRESDEN concept Exzellenz aus Wissenschaft und Kultur

1 DOCUMENT HISTORY

Version	Datum	Autor	Kommentar
001	2016-08-08	Stephan Hensel	Integration der abgelehnten SEMANTiCS-Einreichung

Methodology for conflict detection and resolution in Semantic Revision Control Systems

A step towards technology independent semantic merge management

Stephan Hensel
Chair for Process Control Systems Engineering
Technische Universität Dresden, Germany
stephan.hensel@tu-dresden.de

Markus Graube
Chair for Process Control Systems Engineering
Technische Universität Dresden, Germany
markus.graube@tu-dresden.de

Leon Urbas
Chair for Process Control Systems Engineering
Technische Universität Dresden, Germany
leon.urbas@tu-dresden.de

ABSTRACT

Revision control mechanisms are a crucial part of information systems to keep track of changes. It is one of the key requirements for industrial application of technologies like Linked Data which provides the possibility to integrate data from different systems and domains in a semantic information space. A corresponding semantic revision control system must have the same functionality as established systems (e.g. Git or Subversion). There is also a need for branching to enable parallel work on the same data or concurrent access to it. This directly introduces the requirement of supporting merges.
This paper presents an approach which makes it possible to merge branches and to detect inconsistencies before creating the merged revision. We use a structural analysis of triple differences as the smallest comparison unit between the branches. The differences that are detected can be accumulated to high level changes, which is an essential step towards semantic merging. We implemented our approach as a prototypical extension of the revision control system R43ples to show proof of concept.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database ApplicationscategoryH.3.3Information Storage and RetrievalInformation Search and Retrieval; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Information networks*; H.4 [**Information Systems Applications**]: Miscellaneous

Keywords

Conflicts, Differences, Linked Data, Merging, Named Graphs, Query, Revision, SPARQL, Versioning

SEMANTiCS '16 Leipzig, Germany

DOI: 10.1145/1235

1. INTRODUCTION

The use of Linked Data technologies offers the possibility to integrate data from different systems and domains into a semantic information space. Several problems must be considered regarding client-side access to this information space, for example a restricted bandwidth or a temporary deficiency of the network connection. A common approach is the replication of data, so that a client can work without being dependent on a network connection. However, mobile clients only have limited capacity for storing and processing, so that local replication of the whole data set is not feasible [13]. Only necessary data is queried, mostly by using SPARQL queries against a corresponding triple store. After the local data has been changed, synchronization must follow, taking into account the possibility of concurrent changes by other clients. This raises the need for change traceability. Revision control systems have not been integrated into the semantic web very well [4]. Beside the revision control tasks, conflict detection and resolution in a merge is a crucial part of such a system. The need for human intervention should be minimized or the best possible assistance provided.

2. MOTIVATION

2.1 Revision Control for Triples

Revision control for triples is derived from current revision control systems like Git[1], Subversion[2], CVS[3] or mercurial[4]. The main difference is that triple sets do not have line ordering. However, the functionalities are very similar. A semantic web revision control system should also support data access through revisions. Each revision is identified by a unique identifier referencing an unambiguous state of the repository. Branches which are used for parallel development and the merge of those diverged branches are features to provide too. For example, a merge is required if various clients starting from an initial revision add different features. After they have finished, their changes should be incorporated in the final revision [18]. Such revision control systems

[1] http://git-scm.com/
[2] https://subversion.apache.org/
[3] http://www.cvshome.org/
[4] https://www.mercurial-scm.org/

can enhance the traceability of existing approaches like for example [21].

2.2 Related Work

Redmond et al. [11] presented the initial design and corresponding requirements of a version control system for a Web Ontology Language (OWL). The authors introduced different conflict management possibilities. In general the concurrent access on data can be divided into two main techniques which are either optimistic or pessimistic. In the following we give a brief overview of the main aspects of these approaches and their use in current work regarding semantic revision control systems.

2.2.1 Pessimistic approaches

Pessimistic approaches use a *Lock-Modify-Write/ Unlock* mechanism. Looking at file based version control, a user first has to lock all the files to be changed. These can then be modified and when the modification is finished the locks must be released. A user is not able to change a file while another user has a lock on it. So there is no need for merging of diverged versions.

There are several approaches in literature using such a pessimistic strategy. For example Scheglmann et al. [14] proposed a locking-based approach which allows the simultaneous execution of independent transactions against an ontology. Transactions which would influence other transactions are blocked. Seidenberg et al. [15] had provided a suitable methodology. The strategy which is presented in the corresponding paper aims to mitigate errors in a multi-user ontology editing environment through the usage of locks.

Some drawbacks arise when using a pessimistic strategy. A user may have to wait to execute changes until another user releases the locks. If there is a deficiency of the network connection a user will not be able to release locks on data or get new locks in order to modify data. We want to consider situations where central repository access is not granted but nevertheless the user should have the possibility to modify the data.

2.2.2 Optimistic approaches

Optimistic approaches use a *Copy-Modify-Merge* mechanism. Looking at file based version control, users can simultaneously change files, which have to be merged afterwards. The merge can either be done manually or (semi-)automatically. The advantage of the independent modification of data comes with an increased complexity when a merge must be executed. Merging needs the knowledge of differences and conflicts which may result between two revisions. Below we introduce some current work which uses optimistic approaches to permit revision control in the semantic web context or provide technologies for difference detection.

SemVersion is the implementation of an Resource Description Framework (RDF)-centric approach for structurally and semantically revising RDF models and RDF-based ontology languages (e.g. Resource Description Framework Schema (RDFS)). The revising takes place at the RDF level. With the help of the additional definition of an ontology level semantic diffs can be supported. Blank Nodes are facilitated too through a so called *blank node enrichment* those nodes are unambiguously identifiable [19]. Delta is an ontology describing differences between RDF graphs. The related paper shows the problem of comparing different RDF graphs. It describes how differences can be generated and how a graph can be updated using them. Differences are distinguished in a *weak patch* (context-sensitive) and *strong patch* (context-free) [1]. Schandl [13] introduces the replication of RDF graph parts on (mobile) clients for which computing power, network connectivity and storage capacity are limited. In offline phases, the client therefore works on local replicated sub graphs before submitting changes in the next online phase. A mobile semantic collaboration platform is presented by Ermilov et al. [2]. OntoWiki Mobile is based on the OntoWiki framework extended by advanced replication and conflict resolution mechanisms for RDF content. The approach is partly based on the EvoPat method presented in [12]. It uses patterns for the evolution and refactoring of knowledge bases. In [10] and [9] a language $\mathcal{L}$ is proposed which allows the description of high level changes in RDFS. A corresponding change contains conditions which have to be fulfilled to detect a high level change. As a result the authors identified 132 different changes, capturing modifications like for example additions, deletions or renamings.

Papavasileiou et al. [9] said that revision control systems should generate deltas in a manner which is interpretable for humans and machines. Machine processing demands that changes are described in a consistent and deterministic manner. Each change correspond with exactly one delta. The interpretability by humans includes that single changes are depicted intuitive and concise reflecting the intention of a change (see also [6] and [16]).

The approaches mentioned above are partly technology dependent which means the concept works for example only with RDFS or there are only diffs mentioned without looking at merge management. Overall there is no common conflict detection and resolution strategy.

2.2.3 Conflict detection

Lippe et al. [7] distinguish between two kinds of conflict detection. *State-based merging* use only initial and final states and omit the rest of the revision information. It is fast and easy to implement. However, there are several disadvantages, for example the possibility of detecting unnecessary or incorrect conflicts. Also it is possible that conflicts cannot be detected. In contrast, *operation-based merging* analyses conflicts between revisions on the basis of the operations performed. This offers advantages such as the detection of unnoticed conflicts in a state-based process and more adequate merge results. It provides more support for conflict resolution and is more consistent from the point of view of the user.

Semantic web revision control systems revise triple sets which have no internal line ordering. Thus, structural diffs are generated by the set-theoretic difference of the corresponding RDF triple sets [19]. An essential aspect of merging is the possibility to automatically resolve conflicts and to provide meaningful support to a user when doing a manual merge. Operation-based merging will better support the desired features because of the advantages described and the more general approach.

2.2.4 R43ples

R43ples (Revisions for Triples, pronounced as /'rɪpəls/) [4] is a revision control system which is provided as open

source[5] and can be attached to existing triple stores. It partly bases on the work of van der Sande et al. [18] and extends their concept to fully semantic revision management. The revision information is stored using RDF graphs. Each commit is associated with an *ADD*-set and *DEL*-set which stores the added or deleted triples in a graph. The connections between revisions, corresponding *ADD*- and *DEL*-sets and additional revision information is stored within one RDF graph.

Additional SPARQL keywords have been defined in order to provide the revision management functionality making it possible to query, update, tag and branch different revisions of named graphs. R43ples receives the queries, extracts the revision information and reformulates the queries. Then, it passes them to an existing SPARQL endpoint like Virtuoso[6], Stardog[7] or a Jena TDB[8] database. R43ples itself does not store any information. In general it uses an optimistic strategy for collaborative work because the information stored in a revised graph can be connected with other information spread over the global Linked Data cloud. The usage of pessimistic approaches would imply a locking of the whole network, which is impossible. However, R43ples as presented in [4] does not yet support the merge of different branches in an appropriate manner. In this paper we are presenting an approach to overcome this drawback.

2.3 Contributions

We propose a methodology for conflict detection and resolution in semantic revision control systems as a first step towards technology independent semantic merge management. Existing approaches like [20] do not use the history of changes to compute the delta between two RDF models. Our approach is based on a structural analysis of differences using an operation-based methodology which includes the history of changes to detect further conflicts (definition for conflicts is given in section 3.5.1). The enriched structural data allows the identification of a subset of high level changes analog to [9].

For evaluation purposes, the approach was implemented as an extension of R43ples. Additional SPARQL extensions provide a uniform merge interface and make use of integrated merge models. The automation level of conflict resolution can be specified through different merge methods by the user. The structural conflict analysis is executed on the server side to relieve the client. Furthermore, a client application is implemented to support different levels of detail or information accumulation.

3. APPROACH

3.1 Overall Architecture

The overall architecture of the merge process consists of the interaction of a client and a server. First, the client initiates a Three-Way-Merge. The server receives the merge query and creates revision progresses for both branches starting with the common predecessor as described in section 3.3. This information is used to generate structural differences between the branches which is presented in section 3.4. If there are no conflicts as defined in section 3.5.1 the server merges the branches automatically otherwise the resulting model is sent to the client. The client now has the possibility to enrich the received model structurally and semantically. The manually defined conflict resolution is then sent to the server within a new merge query to execute the merge.

[5]https://github.com/plt-tud/r43ples
[6]http://virtuoso.openlinksw.com/
[7]http://stardog.com/
[8]https://jena.apache.org/documentation/tdb/

3.2 Mathematical background

Basically, a triple consists of a subject, a predicate and an object. So it can be defined as combination of two disjoint infinite sets $\mathbb{U}$ (representing all possible URIs) and $\mathbb{L}$ (representing all possible literals) [10]. Equation 1 presents the mathematical definition of the set of all triples $\mathcal{T}$. To support blank nodes, a prior Skolemization should take place. Merging different sources which include blank nodes would considerably increase the complexity [5]. In this case the junction of revisions would lead to a graph isomorphism problem [1]. Currently the Skolemization must be done by the client as R43ples does not yet support it by itself. This functionality should be added to the server side within future work. Finally, equation 2 defines the set of an RDF graph $\mathcal{V}$ as a subset of $\mathcal{T}$ [10]. Equation 1 and 2 are the fundamental definitions of triple sets. Equation 3 shows some further basic definitions used in this paper.

$$\mathcal{T} := \mathbb{U} \times \mathbb{U} \times (\mathbb{U} \cup \mathbb{L}) \tag{1}$$

$$\mathcal{V} \subseteq \mathcal{T} \tag{2}$$

$$\begin{aligned} &X, X_i\ , X_j^+ \in \mathcal{T} \\ &a, d, n \in \mathbb{N} \end{aligned} \tag{3}$$

3.3 Revision progress creation

3.3.1 *Mathematical description*

The basic definitions for the operation-based approach used to identify structural differences are listed in equation 4. $\mathcal{K}$ describes the set of all four possible states of a triple in a revision. A state identifies the last change of a specific triple related to the start revision of the analysis. The simplest case would be that a not-existing triple can be added and an already existing one can be deleted. But there is also the possibility that an already existent triple was deleted and afterwards added or an not-existent triple was added and later deleted. An example revision graph which includes such changes is later presented in figure 2. The states do not include the alteration time of the triple because the commit time stamp may not be the same as the real time stamp of the triple modification. For example, the client might have no network connection to commit the changes. As the state identifies the last performed operation regarding a specific triple the states $-, 0$ and $+$ are self-explanatory. The last state $\emptyset$ is necessary for the difference model generation described in section 3.4.

$$\begin{aligned} \mathcal{K} := \{ &- = \text{“Deletion of triple”}, \\ &0 = \text{“Triple was not changed since start revision”}, \\ &+ = \text{“Addition of triple”} \\ &\emptyset = \text{“Triple is not included”}\} \\ g, h \in \mathcal{K} \end{aligned} \tag{4}$$

Ω is a set of tuples containing a triple and its corresponding state. In start set Ω_{Start} all triples of the starting revision have the state 0 regardless of their previous history, as presented in equation 5.

$$\Omega_{Start} := \{(X,0) \in (\mathcal{T} \times \mathcal{K}) | X \in \mathcal{V}\} \tag{5}$$

Equations 6 and 7 define the *Add* function and the *Del* function that specify how a triple state can be updated.

$$\begin{aligned} Add(\Omega, X) : \mathcal{P}(\mathcal{T} \times \mathcal{K}) \times \mathcal{T} \rightarrow \mathcal{P}(\mathcal{T} \times \mathcal{K}) : \\ \Omega \mapsto (\Omega \backslash \{(X,-),(X,0)\}) \cup \{(X,+)\} \end{aligned} \tag{6}$$

$$\begin{aligned} Del(\Omega, X) : \mathcal{P}(\mathcal{T} \times \mathcal{K}) \times \mathcal{T} \rightarrow \mathcal{P}(\mathcal{T} \times \mathcal{K}) : \\ \Omega \mapsto (\Omega \backslash \{(X,+),(X,0)\}) \cup \{(X,-)\} \end{aligned} \tag{7}$$

Function R describes the combined execution of the *Add*- and *Del*-function related to the *ADD*- and *DELETE*-set of one revision. The Ω set of the previous revision is required to generate the Ω of the current revision. Equation 8 specifies the execution on one revision with an *ADD*-set with a triples and a *DELETE*-set with d triples.

$$R = Add(...(Add(Del(...(Del(\Omega, X_{0\ }),...), X_{d\ 1}), X_0^+),...), X_{a\ 1}^+) \tag{8}$$

The composition of the separate R-functions takes place along the revision path which contains n revisions. Ω_{Start} designates the set of the starting revision already defined in equation 5. Equation 10 specifies the result of the composition (general definition is exposed in equation 9).

$$(f \circ g)(x) = f(g(x)) \tag{9}$$

$$\Omega_{End} = (R_{n\ 1} \circ R_{n\ 2} \circ ... \circ R_0)(\Omega_{Start}) \tag{10}$$

The generation of Ω_{End} of the branches to be merged makes it possible to detect structural conflicts. The nearest common predecessor of the branches to be merged will be used as the starting revision.

3.3.2 Linked Data Model

To formalize the mathematical description in a semantic way we provide the Merge Management Ontology (MMO). The first part shown in figure 1 describes the revision progress Ω_{End}. The included *RevisionProgress* contains all triple states of one revision path. The definition and the use of the child elements of the object property *statement* are based on [8] as an extension of [17]. The reference between the revision where the status change took place and the triple is defined by another object property for example *rmo:references* (relating to our proof of concept implementation). Further, the triples are stored analogously to [8]. All necessary information is stored in one graph in order to facilitate a subsequent transfer of the data to a client.

Figure 1: MMO visualization of revision progress part (notation of [3])

3.3.3 Example

The example revision graph shown in figure 2 is used to illustrate our approach. It consists of six revisions organized in three branches (MASTER, B1, B2). Each revision has a corresponding *ADD*- and *DELETE*-set. For a simplified illustration only characters are used to represent a triple composed of subject, predicate and object.

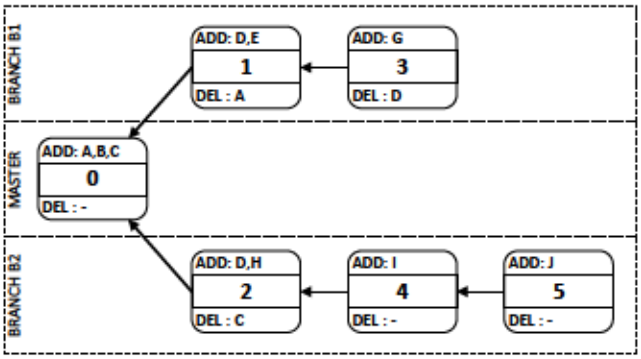

Figure 2: Example revision graph

If we want to merge branch B1 into B2, revision progresses of both branches have to be created. Their common predecessor is revision 0. Table 1 shows the application of the mathematical definitions updating the triple state along the revision path. State changes between two revisions are marked gray. The last column of revision three and five represents the last revision of the corresponding branch and contains the retraced revision path history. These two revision progresses are the starting point for conflict detection and resolution described in the following sections.

3.4 Difference model generation

3.4.1 Mathematical description

The first step is to define differences and derived conflicts. A difference is regarded as a triple contained in both revision progresses with a differing state. The following definitions involve two diverged branches A and B for merging. We assume that the revision progress creation was already finished. Difference detection between the branches takes place. Equation 11 describes the filtering out of all triples without differences. The resulting sets only containing differing triples.

$$\begin{aligned} \mathcal{D}_A &= \Omega_{End(A)} \backslash (\Omega_{End(A)} \cap \Omega_{End(B)}) \\ \mathcal{D}_B &= \Omega_{End(B)} \backslash (\Omega_{End(A)} \cap \Omega_{End(B)}) \end{aligned} \tag{11}$$

The cardinality of $\mathcal{D}_A$ and $\mathcal{D}_B$ is not equal. For this purpose we extend each set with all triples only included in the other set (equation 12) assigning the state of *notIncluded* to them. Afterwards the cardinality of both sets is equal. This is a precondition for the application of equation 13 which is used for the further analysis of differences between the branches to merge.

$$\begin{aligned} \tilde{\mathcal{D}}_A = \{(X,\emptyset) \in (\mathcal{T} \times \mathcal{K}) | (X,g) \in \mathcal{D}_B \wedge (X,h) \notin \mathcal{D}_A; \\ g,h \ arbitrary\} \cup \mathcal{D}_A \\ \tilde{\mathcal{D}}_B = \{(X,\emptyset) \in (\mathcal{T} \times \mathcal{K}) | (X,g) \in \mathcal{D}_A \wedge (X,h) \notin \mathcal{D}_B; \\ g,h \ arbitrary\} \cup \mathcal{D}_B \end{aligned} \tag{12}$$

Triple	States in B1			States in B2			
	Revision 0	Revision 1	Revision 3	Revision 0	Revision 2	Revision 4	Revision 5
A	0	-	-	0	0	0	0
B	0	0	0	0	0	0	0
C	0	0	0	0	-	-	-
D		+	-		+	+	+
E		+	+				
G			+				
H					+	+	+
I						+	+
J							+

Table 1: Triple state update of branches B1 and B2

On the base of the results of equation 12 the intersection of both sets is shown in equation 13. Each triple has two states corresponding to the states of the triple in each branch. The resulting set is the starting point for the difference and conflict detection as described in section 3.5.

$$\mathcal{D}_{Diff} = \{(X, k_A, k_B) \in (\mathcal{T} \times \mathcal{K} \times \mathcal{K})\} | \\ (X, k_A) \in \tilde{\mathcal{D}}_A \wedge (X, k_B) \in \tilde{\mathcal{D}}_B\} \quad (13)$$

3.4.2 Linked Data Model

The second part of the MMO defines the structure of the difference model $\mathcal{D}_{Diff}$ which is composed of the main elements *DifferenceGroup* and *Difference* (see Figure 3). A *DifferenceGroup* groups all differences based on the same triple state combination. The dedicated *Difference* elements specify the triple and the referenced revision where the status change took place. The triples are stored as previously mentioned in RPO. Listing 1 shows an example of a difference model.

```
diff:deleted-added a mmo:DifferenceGroup ;
  mmo:hasDifference [
    a mmo:Difference ;
    mmo:hasTriple [
      rdf:subject ex:testS ;
      rdf:predicate ex:testP ;
      rdf:object "D"] ;
    mmo:referencesA <http://testGraph-rev-3>;
    mmo:referencesB <http://testGraph-rev-2>] ;
  mmo:hasTripleStateA mmo:Deleted ;
  mmo:hasTripleStateB mmo:Added .
```

Listing 1: Part of an example difference model

3.4.3 Example

Based upon table 1 results corresponding to equation 13 are shown in table 2. The last column is explained in the next section.

3.5 Conflict definition

3.5.1 Basic definition

Conflicts are based on triple state combinations. Table 3 shows all possible triple state combinations between two revisions related to one specific triple. All cells of triple state combinations which are not possible by definition are marked black, equal triple states are marked gray, conflicts

triple	state B1	state B2	conflicting
A	-	0	no
B	0	0	
C	0	-	no
D	-	+	yes
E	+	Ø	no
G	+	Ø	no
H	Ø	+	no
I	Ø	+	no
J	Ø	+	no

Table 2: Difference detection and conflict analysis

are crossed and all non-conflicting states are marked white containing the resulting triple state. In general a conflict occurs when the triple was added in one branch and remove in the other one.

Triple state branch A \ Triple state branch B	Ø	-	0	+
Ø		-		+
-	-		-	
0		-		+
+	+		+	

	impossible by definition
	triple states are equal
	conflict detected
+/-	difference detected (cell contains resulting triple state)

Table 3: Triple state table

3.5.2 Conflict handling

On the base of the presented definitions in the previous section a system can merge diverged branches either automatically or semi-automatically. Differences can be solved fully automatically as shown in table 3. For the resolution of conflicts there are at least two possibilities. The first one would be the definition of branch priorities. The triple state of the branch with the higher priority will be the resulting state of the triple in the merged revision. The second one is

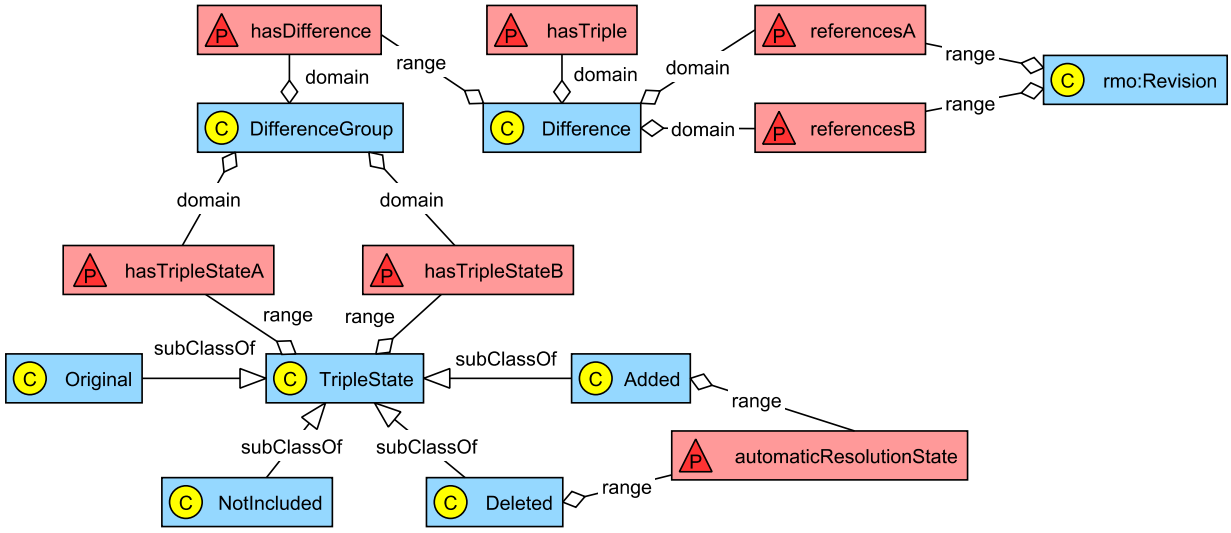

Figure 3: RDO visualization (notation of [3])

use case specific. For example a user has to decide which is the right state of the triple in the merged revision or there are further rules defining it. The following section 4 presents the application of our approach as an implementation extension of R43ples.

4. IMPLEMENTATION

4.1 SPARQL extensions

To test the feasibility of our approach, we implemented it as an extension of R43ples, which already extends SPARQL with additional keywords. Further additions are necessary to support merging. A new merging process is launched by a corresponding query executed against the revision control system. For this, a new keyword `MERGE` is defined. The user can merge the two leafs of different branches which are identified by the branch names separated with `INTO`. The branch name which follows after `INTO` declares the branch on which the merged revision should be created.

The merge query starts with the definition of a user and a message describing the current change. The corresponding graph of the merge also has to be defined. After receiving the query, the server collects and enriches all necessary information to perform the merge process. At the beginning, the nearest common predecessor revision is searched in the revision graph. The revision progress is generated then with the help of the MMO. The result of the comparison is stored in a difference model which is also based on the MMO. The following presentation of the different merge concepts is based on the assumption of a completed model generation process.

4.1.1 MERGE query

Listing 2 shows a merge query. The query flow depends on the difference detection. If there are no conflicts an automatic merge can take place and as a result of the query the user receives the revision number of the merged revision. Otherwise no revision is generated and the user gets the difference model in an RDF serialization.

```
USER "shensel"
MESSAGE "Merge branches."
MERGE GRAPH <graphURI>
  BRANCH "B1" INTO "B2"
```

Listing 2: MERGE query

4.1.2 MERGE query specifying conflict resolution

This query should be used after a MERGE has returned the difference model in case of a detected conflict. The client resolves all conflicts with the help of this model. The result leads to a new MERGE query which is shown in listing 3. Based on the dependency between difference model and rejected merge query, it must be ensured that the same revisions are referenced. The branch names are not enough for this, because another client could have made changes which would result in a false merge. To avoid this problem and to ensure the statelessness of the REST service, it is vital to use concrete revision numbers on which the current difference model is based. The server now has the possibility to reject queries which are executed against an obsolete data status.

In addition to the merge query, the user can define all triples out of the set of the conflicting triples which should be in the merged revision within the `WITH` part. Triples which are not included in this set are ignored and are not included in the resulting revision. All non-conflicting differences are resolved by the definitions in table 3. After the query has been successfully executed, the client receives the revision number of the merged revision.

```
USER "shensel"
MESSAGE "Merge branches."
MERGE GRAPH <graphURI>
  BRANCH "28" INTO "42" WITH { ... }
```

Listing 3: MERGE query with conflict resolution specification

4.1.3 Automatic MERGE query

The keyword `AUTO` allows a fully automatic merge. Possible conflicts will be resolved with the help of branch priorities. The branch specified in front of the keyword `INTO` indicates the resulting state. Listing 4 shows the structure of this query which, after execution, returns the merged revision number.

```
USER "shensel"
MESSAGE "Merge branches."
MERGE AUTO GRAPH <graphURI>
  BRANCH "B1" INTO "B2"
```

Listing 4: Automatic MERGE query

4.1.4 Manual MERGE query

The keyword `MANUAL` is used like `AUTO`. On the server side, all triples specified in the `WITH` part of the query are used as the whole revision content of the merged revision. In listing 5 the structure of a manual merge is shown. This query type is needed if the user not only changes conflicting triples in the difference model. In this case all triples of the resulting revision must be specified.

```
USER "shensel"
MESSAGE "Merge branches."
MERGE MANUAL GRAPH <graphURI>
  BRANCH "B1" INTO "B2" WITH { ... }
```

Listing 5: Manual MERGE query

4.2 Server/Client

The merge concept has been implemented as an extension of R43ples and was tested with Virtuoso (version 7.0) and Stardog (Version 4.0) as attached triple store. The corresponding version of R43ples is open source and can be found at https://github.com/plt-tud/r43ples/releases/tag/v0.8.8. The implementation adds further information to the difference model additional to the mathematical definition. The additional information helps the user to identify the correct conflict resolution with the help of a reference to the revision where the state change took place. The calculation of this information would need more resources than a server-side generation within the structural difference analysis.

In addition to the server extensions, there are currently two implementations which are supporting the user to apply the improved merge functionalities. The web view provided by the server and the additional stand-alone client implementation[9] are offering a convenient method to execute merges. In addition to the merge query generation the implementations are providing three different views to support the user resolving conflicts. These views show different levels of detail or information accumulations. The first view only shows structural differences on triple level. Secondly, a view of individuals is provided which compares the whole contents of the revisions to be merged at the level of RDF individuals. The comparison is based on the RDF property *rdf:type*. The last view enriches the structural difference information on a high level in order to identify changing literals.

[9]https://github.com/plt-tud/r43ples-merging-client/releases/tag/v1.0.0

4.3 Performance evaluation

The current approach generates the revision progress for each branch when it is needed. This works very well for small graphs but for huge graphs where already the common predecessor revision contains a lot of triples this revision progress creation has some drawbacks. For the revision progress creation we use temporary graphs which are filled with content through SPARQL queries. These are mainly COPY, INSERT and DELETE queries. A test shows that the system needs 9.7 minutes to create Ω_{Start} of a revision which contains about 1.5 million triples (Stardog was used as attached triple store). The performance of the further revision progress creation along the revision path of the branch depends on the size of the change sets and the length of the revision path.

A possible solution to overcome the performance drawbacks is to store a revision progress for each branch which is updated when a new commit to the branch is executed. If a client initiates a merge the revision progresses are already generated which will increase the performance of the system significantly but will lead in a higher amount of storage.

5. DISCUSSION

The implementation of the approach presented here allows a client to execute merge queries by uniform interfaces. The server-side generation of the difference and conflict analysis helps small clients with limited performance to process a visualization and to generate the result.

In the current approach the automatic merge uses implicit branch priorities. Merge definitions are only based on a structural difference analysis at the triple level. A semantic enrichment simply takes place on the client side. In particular small clients do not always have the performance to execute such an enrichment. Here, it could be better to perform a pre-analysis on the server side for a better user support. Afterwards it could be possible to merge at a semantic level when the semantics are included in the difference model. For example, a priority based merge at the level of individuals would be possible. Thus, specific individuals may be preferred and all their corresponding triples will be included in the merged revision. Simple high level changes are detectable independently from the vocabulary used. Complex high level changes, e.g. changes of subclasses, are not detectable in such a general approach. Further information is needed to identify changes at this abstraction level.

In future work we will enrich structural diffs with information dependent on the vocabulary. This offers the possibility of full semantic merges. For example sub class changes would be identifiable and the whole differences and conflicts detection can also take place at a semantic level. Furthermore, we want to apply alternative merge approaches like Fast-Forward or Rebase (both well established in Git) to make the merge process more efficient. With the provided web view we want to evaluate our approach with a wide user base and real data to get further results regarding performance and usability.

6. CONCLUSIONS

This paper presents an approach of the integration of merges in semantic revision control systems. Through the definition of SPARQL extensions and models, a uniform interface was created to initiate merges and execute conflict

resolutions.

The advantage is the use of an operation-based approach including the history of changes to detect further differences and corresponding conflicts. Merges can be performed at different automation levels. So the user is very free in the execution of the merge but is supported as far as possible by a preliminary server-side analysis. The implementation of our concept as an extension of R43ples shows the applicability in an already existing system. At once it was shown that a semantic enrichment is possible to detect high level changes in a general approach with the help of the client implementation which uses the generated interfaces.

As already discussed in section 5 our approach still needs further research to enrich the server-side analysis semantically and in the area of semantic merging as well as alternative merge methods.

7. REFERENCES

[1] T. Berners-Lee and D. Connolly. Delta: an ontology for the distribution of differences between RDF graphs., 2004.

[2] T. Ermilov, N. Heino, S. Tramp, and S. Auer. Ontowiki mobile – knowledge management in your pocket. In *The Semantic Web: Research and Applications*, volume 6643 of *Lecture Notes in Computer Science*, pages 185–199. Springer, 2011.

[3] S. Goyal and R. Westenthaler. RDF Gravity (RDF Graph Visualization Tool). http://semweb.salzburgresearch.at/apps/rdf-gravity/user_doc.html, 2004.

[4] M. Graube, S. Hensel, and L. Urbas. R43ples: Revisions for Triples - An Approach for Version Control in the Semantic Web. In *Procedings of Linked Data Quality (LDQ) Workshop*, Leipzig, 2014.

[5] T. Heath and C. Bizer. *Linked Data: Evolving the Web Into a Global Data Space*. Synthesis Lectures on Web Engineering Series. Morgan & Claypool, 2011.

[6] M. Klein. *Change management for distributed ontologies*. PhD thesis, Vrije University, 2004.

[7] E. Lippe and N. van Oosterom. Operation-based Merging. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, SDE 5, pages 78–87, New York, NY, USA, 1992. ACM.

[8] Open Knowledge Foundation. Change Sets. https://pythonhosted.org/ordf/ordf_vocab_changeset.html, 2010.

[9] V. Papavasileiou, G. Flouris, I. Fundulaki, D. Kotzinos, and V. Christophides. High-level Change Detection in RDF(S) KBs. *ACM Trans. Database Syst.*, 38(1):1:1–1:42, 04 2013.

[10] V. Papavassiliou, G. Flouris, I. Fundulaki, D. Kotzinos, and V. Christophides. On Detecting High-Level Changes in RDF/S KBs. In *The Semantic Web - ISWC 2009*, volume 5823 of *Lecture Notes in Computer Science*, pages 473–488. Springer, 2009.

[11] T. Redmond, M. Smith, N. Drummond, and T. Tudorache. Managing Change: An Ontology Version Control System. In *OWLED*, 2008.

[12] C. Rieß, N. Heino, S. Tramp, and S. Auer. EvoPat – Pattern-Based Evolution and Refactoring of RDF Knowledge Bases. In *The Semantic Web – ISWC 2010*, number 6496 in Lecture Notes in Computer Science, pages 647–662. Springer, 2010.

[13] B. Schandl. Replication and Versioning of Partial RDF Graphs. In *The Semantic Web: Research and Applications*, volume 6088 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2010.

[14] S. Scheglmann, S. Staab, M. Thimm, and G. Gröner. Locking for Concurrent Transactions on Ontologies. In *The Semantic Web: Semantics and Big Data*, number 7882 in Lecture Notes in Computer Science, pages 94–108. Springer, 2013.

[15] J. Seidenberg and A. Rector. A Methodology for Asynchronous Multi-user Editing of Semantic Web Ontologies. In *Proc. of the 4th Int. Conf. on Knowledge Capture*, pages 127–134. ACM, 2007.

[16] L. Stojanovic. *Methods and tools for ontology evolution.* PhD thesis, Karlsruhe Institute of Technology, 2004.

[17] S. Tunnicliffe and I. Davis. Changeset. http://vocab.org/changeset/schema.html, 2005.

[18] M. Vander Sande, P. Colpaert, R. Verborgh, S. Coppens, E. Mannens, and R. Van de Walle. R&Wbase: git for triples. In *Proceedings of the 6th Workshop on Linked Data on the Web*, 2013.

[19] M. Völkel and T. Groza. SemVersion: An RDF-based Ontology Versioning System. In *Proc. of IADIS Int. Conf. on WWW/Internet*, pages 195–202, 2006.

[20] D. Zeginis, Y. Tzitzikas, and V. Christophides. On the Foundations of Computing Deltas Between RDF Models. In K. Aberer, K.-S. Choi, N. Noy, D. Allemang, K.-I. Lee, L. Nixon, J. Golbeck, P. Mika, D. Maynard, R. Mizoguchi, G. Schreiber, and P. Cudré-Mauroux, editors, *The Semantic Web*, number 4825 in Lecture Notes in Computer Science, pages 637–651. Springer Berlin Heidelberg, 2007. DOI: 10.1007/978-3-540-76298-0_46.

[21] J. Ziegler, M. Graube, and L. Urbas. RFID as universal entry point to linked data clouds. In *RFID-Technologies and Applications (RFID-TA), 2012 IEEE International Conference on*, pages 281–286, Nov 2012.